EXHIBIT A

Simulation Reference Markup Language

White Paper

Abstract
This white paper describes the application of XML technologies to the area of simulations by introducing the Simulation Reference Markup Language (SRML) and its corresponding runtime environment. SRML is an application of XML for describing simulation models, and its runtime environment is software that is capable of executing those models. The goal of SRML is to enable simulations to be served, received, and processed in a standard fashion using Internet technologies and the World Wide Web, just as HTML enables that functionality for text, and MathML[1] enables that functionality for mathematics. SRML can be used to encode both the structure and behavior of all items comprising a simulation. A small number of SRML tags and pre-designated attributes are used to describe abstract items, while rich internal and external structures are further described and validated by the native rules of XML and simulation-specific schemas.

This white paper is intended primarily for those evaluating the use of XML as a means for describing and executing computer simulations. It is not a specification or User's Guide but rather an introduction and a brief tutorial.

This document begins with background information on the representation and development of simulations in computers, the problems they pose, and the philosophy underlying the solutions SRML proposes. A brief introduction tutorial on the concepts and use of SRML follows. Additional sections describe the runtime environment, as it exists in a software engine known as a Simulation Reference Simulator. The document concludes with a language reference, schema definition, and a brief treatise on simulation tools.

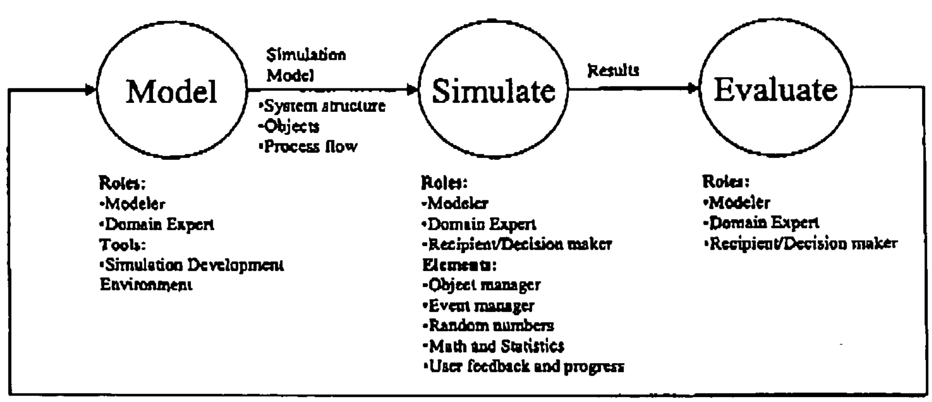
1. Background

Simulation is a process that attempts to predict aspects of the behavior of some system by creating an approximate (mathematical) model of it. Computers provide an ideal environment for building simulations. Simulations have many uses as follows:

- Businesses use simulations to develop and optimize processes.
- Simulations lower risks associated with critical decisions.
- Engineers use simulations to prototype and test designs.

2. General Simulation Process

Generally, simulation follows a three-step process that follows the sequence Model, Simulate, and Evaluate. The process involves several roles: Modeler, Domain Expert, Recipient/Decision maker. Modelers and domain experts build models using a tool known as simulation development environment. This tool generates and stores a representation of the model that is fed into a simulation engine that manages objects, events, math, statistics and random numbers. The engine typically provides some kind of feedback about the simulation progress, as running a simulation may take more than a few seconds. Once the simulation has completed, modelers, domain experts, and decision makers evaluate the results and feed those results into the construction of a new model. Thus, simulation usually follows an ongoing iterative methodology. The following diagram illustrates the basic simulation process:



3. Simulation needs and challenges

- Because simulation involves programming, both have similar requirements.
- Common needs: low cost, easy to program, platform independent, reusable, standards-based, graphics-based development environment, graphics based output.
- A simulation model typically needs to represent large numbers of things in relationship to each other.
- Simulation is a subset of programming, and as such poses challenges of its own.
- Simulations need to describe large numbers of objects, lots of events, complex logic.
- Simulations typically require lots of computing speed and memory.

4. Characteristic simulation challenges

- Additional challenges arise in simulations because they can have many characteristics.
- Discrete event (Monte Carlo methods, random events, probabilities) vs. non-discrete
- Distributed vs. individual
- Parallel vs. Serial processing
- Single run vs. continuous execution
- Interactive vs. batch
- General vs. specific

5. XML technologies are well suited to both represent and execute simulations

- XML has enough expressiveness to describe data objects at various levels of complexity.
- Through the use of schemas, XML provides structural document validation and data-typing for free.
- XML can be easily transported in a human readable form to various places.
- XML can easily describe hierarchical structures and linked networks.
- Once loaded, XML data can be queried in a fashion similar to a database.
- Editors exist and are inexpensive or free.
- Tools that read and write XML are free.
- Scripting can be embedded within XML to add behavior.
- XML tools can easily support a simulation runtime environment and are free.

6. XML Background

XML consists of elements that are specified using tags (words bracketed by '<' and '>'). Elements can have both attributes (described by a name-value pairs), and can contain embedded elements:

```
<elementname1 attributename1='value1' attributename2='value2' ...>
        <embedded-element1 ...>
                ...
        </embedded-element1>
        ...
</elementname1>
```

These constructs form a structure for describing any kind of hierarchically related discrete data. A Schema permits a modeler to specify rules about element relationships and attributes that can be validated with software. The XML Document Object Model (DOM) is an Application Programming Interface (API) that enables programming languages and scripts to load, validate, navigate, modify and save XML structures with code. Software libraries and components for the DOM are freely available from several sources, including Microsoft. DOM software implements elements and attributes with linked Node objects. For example, the previous XML construct can be represented in the DOM as:

```
Node (NodeName='elementname1')
        Attribute (BaseName='attribute1', NodeTypedValue='value1')
        Attribute (BaseName='attribute2', NodeTypedValue='value2')
        Node (BaseName='embedded-element1')
```

Use of plain XML does not imply semantic interpretation, presentation, or behavior of the data. Rather, schemas and higher-level specifications, such as Semantic-web for semantics, and XSLT for presentation provide those. Typically behavior comes from other markup language like DHTML and XHTML. However, those languages couple presentation and behavior more tightly than is practical for large-scale simulations.

7. Introduction to the Simulation Reference Modeling Language

The Simulation Reference Markup Language (SRML) is an XML application that provides a means for describing a simulation. An SRML Simulator is a software tool that reads XML input that conforms to the SRML Schema, builds script-based items corresponding to each of the elements and executes events scheduled by those items. An SRML modeling tool is a software program that allows a modeler to create and edit SRML simulations.

SRML organizes XML in a way that makes it practical to represent:

- Large numbers of interconnected items, both hierarchically and networked.
- Individual item behavior via scripts.
- Item classes, polymorphism, and multiple-inheritance.
- A means for synchronous, asynchronous and scheduled communication.
- Random events.

SRML, like XML, does not specify any data presentation. Other markup languages and tools like DHTML, XHTML, Java, Microsoft Office and Visio satisfy that need. XML also does not specify the use of any particular simulation development tools.

8. SRML Tutorial

This section provides a brief tutorial on the primary concepts in SRML.

SRML Language Concepts

The things you describe when modeling in SRML are referred to as items. An item can represent a physical thing, such as a piece of equipment, or a person, or an entire system of other items. An item may also represent a process or a step in a process. SRML provides a natural way to express these items, and complex relationships that may exist among them, using the grammar of XML with a small set of pre-defined elements and attributes that have specific meanings and rules. While XML uses the terms 'element' and 'attribute', SRML uses parallel terms: 'item' and 'property'. The general structure of a simulation in SRML looks like this:

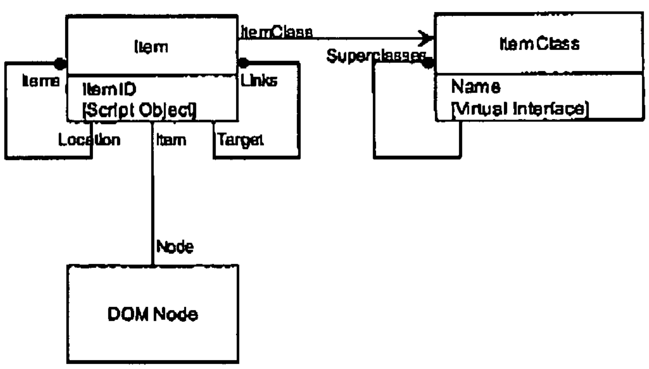
```
<Simulation>
    <!-- Main Script - the following element defines the behavior for the entire simulation -->
    <Script Language='language'>
    code
    </Script>
    <ItemClass Name='itemclassname1' SuperClasses='itemclass1,itemclass2',...' attribute1='value1'>
        <!-- The following element defines the behavior for the class; not individual instances -->
        <Script Language='language'>
        code
        </Script>
        <!-- The following element defines the prototype for the instances -->
        <itemclassname1 attribute1='value1', ...>
            <!-- The following element defines the behavior for the instances -->
            <Script Language='language'>
            code
            </Script>
        </itemclassname1 >
    </ItemClass>
    <ItemClass Name='itemclassname2' ...>
        ...
    </ItemClass>
    <System>
        <itemclassnameA Name='itemname1' Quantity='number' attribute1='value1', ...>
            <Link Name='linkname1' Target='query' />
            <Link ... />
            <Links Name='linksname1' >
                <Link Name='linkname2' Target='query'>
                ...
```

```
                    </Links>
                    <Script Language='language'>
                    code
                    </Script>
                    <itemclassnameB Name='itemname'>
                        ...
                    </itemclassnameB >
            </itemclassnameA>
            <itemclassnameC ...>
                ...
            </itemclassnameC>
        <System>
</Simulation>
```

Items, attributes, and properties

You won't find a specific element called 'item' in the previous construct because the structure implies items by the elements you define. With the exception of a few pre-defined names, such as ItemClass, SRML allows you to define your own elements. Having this flexibility makes it easier for a modeler to create a domain-specific XML schema for validating a system's structure and to provide data types for element attributes. A domain-specific schema works in conjunction with SRML when specifying XML namespaces. The ItemClass element allows you to generalize groups of common items, yet an item does not need to have a corresponding ItemClass. An item's attributes correspond to properties.

The following illustration shows a simplified conceptual model of an item. Internally an item has a unique system-generated ItemID and a script. It has an association with an XML Document Object Model (DOM) node; it can both serve as a location, and belong at a location along with other items; it can have links to other items, and be a target for a link; it can also belong to an item class, which in turn can have super-classes. An item gets its properties from DOM attributes, and its behavior from a script.



The following example shows the definition of a Vehicle element and Name attribute in XML, which corresponds to a Vehicle item that has a Name property with a value of 'Challenger', and a VIN of 'N3462983':

```
<Vehicle Name='Challenger' VIN='N3462983'>
</Challenger>
```

If no vehicle attributes were important, the same structure could be represented legally in SRML as:

```
<Vehicle />
```

Item behavior

You provide an item's behavior inside a Script element using either JavaScript or VBScript, and specifying the Type attribute as either 'text/javascript' or 'text/vbscript'. For example:

```
<Vehicle ID='AirforceOne' Running='1'>
```

```
        <Script Type='text/javascript'>
        <![CDATA[
        var Occupied=0;
        function turnOn()
                {
                Running=1;
                }
        function turnOff()
                {
                Running =0;
                }
        ]]>
        </Script>
</Vehicle>
```

The sample demonstrates several concepts. First, you can think of an item as an object that is defined with an XML element and a script. The prior example defines the Vehicle item with a unique system-wide identifier (the ID attribute has global semantics) and a Running property with an initial value of '1'. In general, the script has direct access to any element attribute. It can also define properties that don't have corresponding attributes by declaring public variables—as in the Operable variable. Third, note that the item's behavior comes by way of functions, such as 'turnOn', and 'turnOff'.

Because the script defines AirForce1 as having a global identifier, anywhere in the script you could use code like this:

```
AirForce1.Occupied=1;
AirForce1.turnOn();
X=AirForce1.Running;
AirForce1.turnOff();
```

Locations
A small number of intrinsic properties apply to all items. For example, every item has a Location property that refers to its current location. In the following sample, the sensor identified by 'TSM5865' has its location set to the controller identified by 'TS293847'. Most of the time an item's location refers to another item, however sometimes an item's location can be 'Nothing' or 'undefined', which means that it either does not exist in the model or is at the top of the location tree. Each item also has an Items collection that holds references to all of the items it contains. In the next example, the controller sends queryStatus to its items when it is asked to poll. When a sensor's queryStatus method is invoked, it invokes the receive method at it's location.

Controller.xml:

```
<Simulation Name='ControllerSimulation'>
        <Controller Name='TS293847' Operable='1' Pings='0' Health='1'>
                <Script Type='text/javascript'>
                <![CDATA[
                function poll()
                        {
                        Items(1).queryStatus();
                        Items(2).queryStatus();
                        }
                function receive(Name, value)
                        {
                        Health=math.min(Health, value);
                        Pings++;
                        }
                ]]>
                </Script>
                <Sensor Name='TSM5865' Quantity='2'>
                        <Script Type='text/javascript'>
                        <![CDATA[
                        function queryStatus()
                                {
                                Location.receive(Name, Random());
                                }
```

```
                    ]]>
                    </Script>
            </Sensor>
        </Controller>
</Simulation>
```

Item quantities

A situation may arise when a large quantity of a particular item or structure needs to occur. Within any element, you can provide the Quantity attribute, which instructs an SRML processor to duplicate the element and its contents. For example, the following structure defines a total of 240 eggs, in a total of 20 egg cartons in two refrigerators.

```
<Refrigerator Quantity='2'>
        <EggCarton Quantity='10'>
                <Egg Quantity='12' />
        </EggCarton>
</Refrigerator>
```

Links

XML provides an ideal structure for describing hierarchical containment relationships; however, situations may arise when items at various locations in a simulation need to have non-hierarchical connections to other items. Therefore SRML provides two linking mechanisms. The first involves using the Link element where you provide the name and an XSL query to identify the target. The second mechanism uses the Links element where you provide a name and supply inner links.

Additional item characteristics

Besides the Location and Items properties, items have other built-in properties. When the SRML simulator initially creates an item, it automatically assigns a unique numeric value to the item's 'ItemID' property. You can use this property to uniquely identify the item in the presence or absence of an explicit ID attribute. Items also have a language-neutral 'Self' property that refers to the particular item instance in which a script is running--however the traditional 'Me' (VBScript) and 'this' (JavaScript) keywords serve the same purpose. An item's Events collection contains references to all the events scheduled or posted for that item. This makes it easy at runtime to find and possibly alter previously scheduled events. As a simulation runs, the scripts operate directly on the underlying XML Document Object Model Node objects as they alter item properties and locations. Therefore each item has a Node property that returns its corresponding node. In addition, each item also contains a reference to its item class via its ItemClass property. You can add shared properties and behavior to a specific item class in the same way you add them to regular items.

Moving items

The static structure of a model represents interconnected items at a point of time. As a simulation progresses, you may need to move an item to a different location. To do this, you simply assign its location property to a different item. With JavaScript, the following code moves Vehicle99 to Alaska.

```
Vehicle99.Location = Alaska
```

Items with fixed locations

Some items, such as geographical locations, may have their locations fixed for an entire simulation. When these items have unique names within their containing locations, such as California within United States, you can make them directly accessible by specifying LocationFixed='True' attribute. Refer to the following example:

Sites.xml:

```
<Location ID='World' LocationFixed='True'>
        <Location Name='USA' LocationFixed='True'>
                <Location Name='California' LocationFixed='True'>
                        <Location Name='Anaheim' LocationFixed='True'>
                                <Vehicle Name='Truck' Quantity='100' />
                        </Location>
                </Location>
        </Location>
</Location>
```

Somewhere in a script, you can address the first Truck in Anaheim within the World by the following syntax:

```
World.USA.California.Anaheim.Items(1)
```

Finding items

You may also need to find items that meet certain criteria. To do so, you can provide an XSL Query string to the FindItems function and it will return a collection of items that were found. For example, this code will set the objSystem variable to the first item found in a location that has its name set to System:

```
objController=FindItems("//Controller[@Name='TS293847']")(1)
```

Events

Typically the items we model correspond to physical things, which include people, the weather, the ocean, the planets, mechanical devices and electronic circuits. Items such as these may both send and receive stimulus to and from other items. The SRML runtime environment provides four procedures for sending synchronous, asynchronous and scheduled events among objects. The procedures are SendEvent, PostEvent, ScheduleEvent, and BroadcastEvent. Each of these procedures takes a target object, an event string, and any number of parameters. Send Event sends the specified event immediately to the target object. Post event sends the event asynchronously to the target. ScheduleEvent takes a date-time parameter and sends the event at the specified time.

Main script

In a typical discrete-event simulation, a host of items will generate events autonomously. Since these items are linked to other items, communication proceeds naturally. However, the need typically arises for the simulation to generate metrics such as a count of event of a certain type over a period of time, or perhaps a trace of those same events. In other cases, the course of the simulation may need to change as the result of an event. By placing a Script element directly inside the Simulation object you can create a 'main script' that can intercept every event in the simulation immediately after it occurs. The main script is purely optional; and providing it allows you to describe a completely self-contained simulation within the SRML. In the next example, the main script receives the 'failed' and 'repaired' events of the Motor item, after the motor has received them.

Motor1.xml:

```
<Simulation>
	<Script Language='JavaScript'>
	<![CDATA[
		var nFailures = 0;
		var nRepairs = 0;

		function Motor_failed(objLRU)
			{
			nFailures++;
			}

		function Motor_repaired(objLRU)
			{
			nRepairs++;
			}
	]]>
	</Script>
	<Motor ID='MO293847' Operable='1'>
		<Script Type='text/javascript'>
		<![CDATA[
		ScheduleEvent(this, "failed", DateAdd("h", Random()*100, CurrentTime));
		function failed()
			{
			Operable=0;
			ScheduleEvent(this, "repaired", DateAdd("h", Random()*8, CurrentTime));
			}
		function repaired()
			{
			Operable=1;
			ScheduleEvent(this, "failed", DateAdd("h", Random()*100, CurrentTime));
```

```
                }
        ]]>
        </Script>
    </Motor>
</Simulation>
```

Item classes

While SRML allows you to provide specific attributes and behavior for individual items, it also provides a generalization mechanism that allows you to create classes of items. An item class is analogous to class in object-oriented terms, in that it allows you to describe the attributes and behavior for its instances while also providing a type of inheritance. You can specify any number of item classes in a model by using an ItemClass element and specifying a unique Name attribute. Within the class you also provide an instance prototype by embedding an element with a tag name that matches the Name attribute of the item class. This prototype can have attributes with default values and data types validated by an XML schema. Refer to the general form of an ItemClass:

```
<ItemClass Name='itemclassname1' SuperClasses='itemclass1,itemclass2',...' attribute1='value1'>
    <!-- The following element defines the behavior for the class; not individual instances -->
    <Script Language='language'>
    code
    </Script>
    <!-- The following element defines the prototype for the instances -->
    <itemclassname1 attribute1='value1', ...>
        <!-- The following element defines the behavior for the instances -->
        <Script Language='language'>
        code
        </Script>
    </itemclassname1>
</ItemClass>
```

The highlighted area indicates the prototype item, and the bold text shows the correspondence between the name of the prototype and the name of the item class. The specific example below, which also highlights the prototype item, demonstrates the definition of an item class called Counter. Each of the ten instances created from this class will have a method called Increment and will have its own Count. As an item itself, the ItemClass can also have properties and behavior that will be shared by the instances. The example defines an Instances property for the class that keeps track of the number of individual Counter instances. Any of the Counter instances can access its item class, through its intrinsic ItemClass property. Notice how the item accesses the Instances property in its item class in the code below. In the note below, code within the instance is directly updating the Instances value of the item class.

Counter.xml

```
<Simulation>
    <ItemClass Name='Counter' Instances='0'>
        <Counter Count='0'>
            <Script Type='text/javascript'>
            <![CDATA[
            ItemClass.Instances++; // Note: Increment the item class's value.
            function Increment()
                {
                Count++;
                }
            ]]>
            </Script>
        </Counter>
    </ItemClass>
    <Counter Quantity='10' />
</Simulation>
```

Super-classes

Sometimes a common set of properties and behaviors apply to more than one item class. This situation presents the opportunity create a more general item class that other more specific classes can inherit from. Therefore, SRML provides a capability found in traditional object-oriented languages for defining classes at various levels of generalization. The mechanism in SRML provides polymorphism, multiple-inheritance, and overriding in both item class instances and the item class itself. The main difference between inheritance in SRML and traditional languages is that the activity of walking up an inheritance chain to find an item's method does not exist. Thus overriding a method overrides its complete definition. In the example below, a Motor has two more general super-classes classes: Asset and Operates.

Motor4.xml:

```
<Simulation>
	<Script Type='text/javascript'>
	<![CDATA[
		var nFailures = 0;
		var nRepairs = 0;

		function Motor_failed(objLRU)
			{
			nFailures++;
			}

		function Motor_repaired(objLRU)
			{
			nRepairs++;
			}
	]]>
	</Script>
	<ItemClass Name='Asset' Count='0'>
		<Asset Operable='1'>
			<Script Type='text/javascript'>
			<![CDATA[
			ScheduleEvent(this, "failed", DateAdd("h", Random()*100, CurrentTime));
			function failed()
				{
				Operable=0;
				ScheduleEvent(this, "repaired", DateAdd("h", Random()*8,
			CurrentTime));
				}
			function repaired()
				{
				Operable=1;
				ScheduleEvent(this, "failed", DateAdd("h", Random()*100,
			CurrentTime));
				}
			]]>
			</Script>
		</Asset>
	</ItemClass>
	<ItemClass Name='Operates'>
		<Operates Running='1'>
			<Script Type='text/javascript'>
			<![CDATA[
			function turnOn()
				{
				Running=1;
				}
			function turnOff()
				{
				Running =0;
				}
			]]>
			</Script>
```

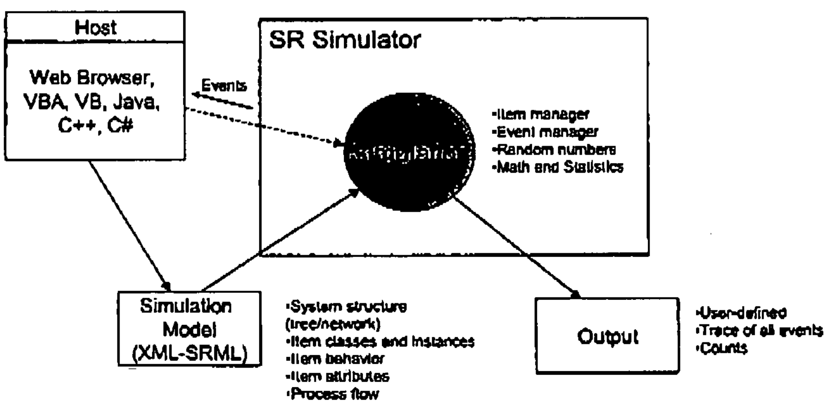
```
            </Operates>
        </ItemClass>
        <ItemClass Name='Motor' SuperClasses='Asset,Operates' />
        <Motor Quantity='10' />
</Simulation>
```

9. Runtime environment

The SRML runtime environment is a collection of software objects that can read SRML input, build and manage the corresponding simulation items, connect those items together, and provide an event-driven mechanism for items to communicate. It also provides simulation support in the form of a random number generator, simulation primitives such as time averages and data structures, math functions, statistics functions, and the ability to generate outputs as defined by the model.

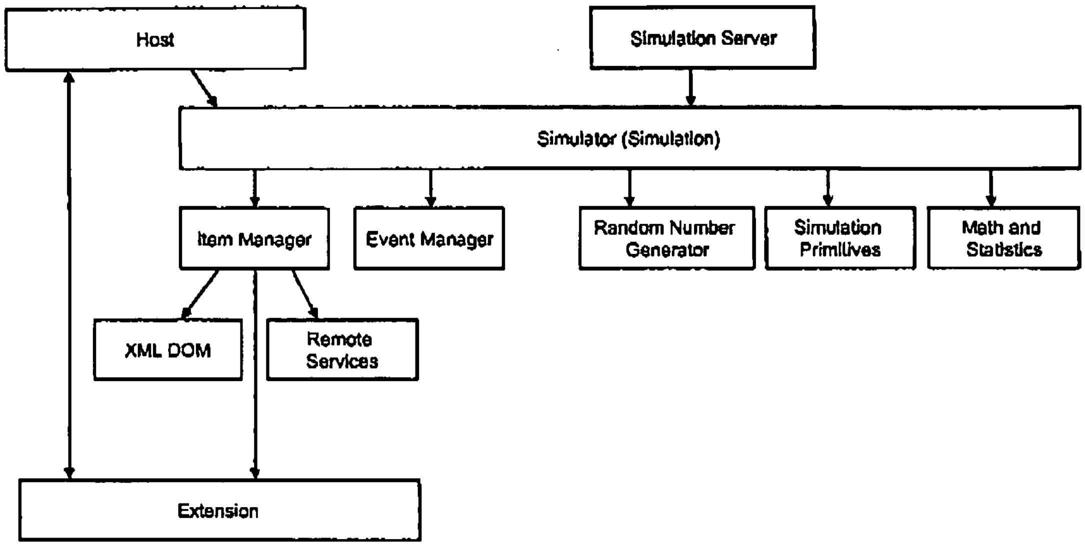
External environment

Simulations run in many external environments. For example, a simulation may run as part of a web page, as part of a spreadsheet, or embedded in a custom application. Thus an SRML simulation can exist an object in component known as the Simulation Reference Simulator (SR Simulator). As a component the simulator requires a host application. A modeler can choose any host that is capable of creating a Simulation object and using its interface. Possible hosts include Microsoft Internet Explorer, Microsoft Office applications, Microsoft Visual Basic, and Java. The following diagram shows a simplified view of how a Simulation object functions within its external environment.



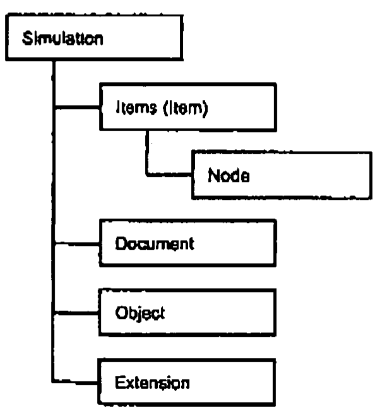
Simulation Object

An SR Simulator provides a class of Simulation objects. Each Simulation object encapsulates a simulation instance (or run), and is primarily responsible for providing the runtime environment. Since a Simulation is an object that embodies a particular execution of a simulation, many simultaneous simulations can exist on the same computer, or distributed across a network in a as a client or server. This next diagram shows internal architecture of a SR Simulator and its associated environment.



Loading a simulation

The simulation loading process begins when a calling procedure invokes the Load method of a Simulation object and passes the location of the XML. At that point, the simulation calls upon an XML parser to build a validated DOM tree from the specified input. If successful, it then traverses the DOM tree and creates items from the nodes, attributes, links, and scripts they contain. Code within an item's script that appears outside a function or procedure gets executed as the item is created. Given that some items are loaded before others, this outer code has limited access to other items. Specifically, an item can only access its Location property, its ItemClass property, any related super-classes, and the Simulation object. Its sub-items, links, and any global items can't be accessed until after the load is complete. If items require further initialization the modeler can broadcast a downward event from the root that invokes a specified initialization method. When the Load has completed the simulation is ready to begin process events, and the Simulation object is configured according to the following object model:



Running a simulation

Once the simulation has been loaded, the Simulation object can execute the model as individual steps (using DoNextEvent, and ForwardLastEvent), or as a single run (using the Run method). In either case, the Simulation object makes every event available to the host by either generating an EventOccurred event or by invoking the method in the host. The sample code below shows how to run a SR Simulator and Simulation from Microsoft Visual Basic:

```
Private WithEvents Sim As Simulation

Private Sub Simulate()
    Set Sim = CreateObject("SR_Simulator.Simulation")
    Sim.RandomizeSeed
    Sim.Load App.Path & "Motor4.xml"
```

```
    Sim.CurrentTime = Now
    Sim.EndTime = DateAdd("yyyy", 1, Simulation.CurrentTime)
    Sim.Run Me, True
    Sim.Print "Failures " & Simulation.Object.nFailures
    Debug.Print "Repairs " & Simulation.Object.nRepairs
End Sub

Private Sub Sim_EventOccurred(objObject As Object, strEvent As String, arrParams As Variant)
    Debug.Print strEvent & " " & Simulation.CurrentTime
End Sub
```

These items read and write to the attributes of their corresponding DOM node as their properties are read and modified.

Additional items to describe:

- Items as objects
- Item manager
- Finding items
- Moving items
- Main script initialization
- Network of interconnected objects
- Event manager
- Simulation object events
- Extensions
- Random number generation
- Statistics: time averaging
- Open architecture
- Queuing and dispatching
- Hosting in VBA
- Hosting with Java
- Other script languages
- Distributing a simulation

10. SRML Language Reference

XML keywords:

Element	Type	Applies	Description
ItemClass	Element	Globally	Designates a class of items, which all have the same behaviors.
JavaScript	Value	Script	Specifies that the script language for an item is JavaScript.
Language	Attribute	Script	Specifies the language that the script uses.
Link	Element	Items	Specifies a link (object reference) to another Item.
Links	Element	Items	Specifies a collection of individual links.
Name	Attribute	Items, ItemClasses	Designates the name for the Item or ItemClass. Name is required for an ItemClass, but not for an Item.
Quantity	Attribute	Items	Specifies the number of items to be constructed at the enclosing location.
Script	Attribute	Items, ItemClasses	Specifies the script, which defines the state and behavior for items.
Simulation	Object	Globally	This is the Simulation object.
SuperClasses	Attribute	ItemClass	Provides an ordered sequence of ItemClasses that provide default behavior for the ItemClass.
Target	Attribute	Link	Specifies the target object of the link.
VBScript	Value	Script	Specifies that the script language is VBScript.
Item	Element	Globally	This is an implied.

Element	Type	Applies	Description
LocationFixed	Attribute	Item	Specifies that an item is fixed within its location. This makes the item directly accessible to the location through either its ID or it's name whichever is specified first. If two items have the same ID or name, the last one will become the item that is directly accessible. For example if itemname is location fixed, then it can be accessed through location.itemname.

Intrinsic Item properties and methods:

Element	Type	Applies	Description
ItemID	Attribute	Item	Returns the unique identifier assigned to the item by the simulation when it was created.
Items	Collection	Item	Refers to the sub-items of the current item.
Location	Attribute	Item	Refers to another item that represents the current location of the item.
Self	Attribute	Item	Returns a reference to the current item. In JavaScript, the keyword 'this' may also be used. In VBScript, the keyword 'Me' may also be used.
Events	Collection	Item	Returns the events currently scheduled for an item.
Node	Attribute	Item	Returns the XML DOM node corresponding to the item.
ItemClass	Attribute	Item	Returns the ItemClass of the item.

Simulation runtime properties and methods:

Element	Type	Applies	Description
Run	Method	Simulation	Runs the simulation object.
Abort	Method	Simulation	Aborts the running simulation.
ScheduleEvent	Method	Simulation	Schedules an event.
CurrentTime	Property (read/write)	Simulation	Sets or returns the simulation's current time.
DestroyItem	Method	Simulation	Destroys an item.
Document	Method	Simulation	Returns the XML DOM document object.
DoNextEvent	Method	Simulation	Sends the next scheduled or posted event to its target.
EndTime	Property (read/write)	Simulation	Sets or returns the simulation's end time.
EventOccurred	Event	Simulation	Indicates that an event occurred in the simulation.
Extension	Property (read/write)	Simulation	Sets or returns the simulation's extension object. This is used to allow items in the simulation to directly access the properties and methods of the extension object.
FindItems	Method	Simulation	Returns a collection of items found using the specified query.
ForwardLastEvent	Method	Simulation	Forwards the last event to the specified object.
GetItem	Method	Simulation	Returns the item with the specified ItemID.
Item	Method	Simulation	Returns the item with the specified ItemID.
ItemCount	Method	Simulation	Returns the number of items in the simulation.

11. XDR Schema for SRML

```
<!-- SRML 1.0 Schema  ....................................................... -->
<!-- file: SRML.xml
-->
<!-- SRML 1.0 Schema
```

```
  This is the Simulation Reference Markup Language (SRML) 1.0, an XML
  application for describing discrete event simulations
  both their structure and behavior.

  Copyright 2001 Steven W. Reichenthal, Boeing

  Permission to use, copy, modify and distribute the SRML 1.0 Schema and
  its accompanying documentation for any purpose and without fee is
  hereby granted in perpetuity, provided that the above copyright notice
  and this paragraph appear in all copies.  The copyright holders make
  no representation about the suitability of the Schema for any purpose.

  It is provided "as is" without expressed or implied warranty.

     Revision:   $Id: $

  Revisions: editor and revision history at EOF
-->
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">

  <AttributeType name='Name' dt:type='string' />

  <AttributeType name='SuperClasses' dt:type='string' />

  <AttributeType name='Language' dt:type='string' />

  <AttributeType name='ItemClass' dt:type='string' />

  <ElementType name='Script' content='mixed'>
    <attribute type='Language' required='yes' />
  </ElementType>

  <ElementType name='ItemClass' content='mixed'>
    <attribute type='Name' required='yes' />
    <element type='Script' />

    <!-- Prototype elements may be added here. -->

  </ElementType>

  <AttributeType name='Target' dt:type='string' />

  <ElementType name='Link' content='mixed'>
    <attribute type='Name' required='yes' />
    <attribute type='Target' required='yes' />
  </ElementType>

  <AttributeType name='Quantity' dt:type='i4' />

  <AttributeType name='LocationFixed' dt:type='boolean' />

  <!-- The reference element, Item, is defined here. -->

  <ElementType name='Item' content='mixed'>
    <attribute type='ItemClass' required='yes' />
    <attribute type='Quantity' required='no' default='1' />
    <attribute type='LocationFixed' required='no' />
    <element type='Script' />
  </ElementType>


  <ElementType name='Simulation' content='mixed'>
    <element type='Script' />
    <element type='ItemClass' minOccurs='0' maxOccurs='*' />
    <element type='Item' minOccurs='0' maxOccurs='*' />

    <!-- Specific item elements may be added here. -->

  </ElementType>

</Schema>
```

```
<!-- Revision History:

       Initial draft 2001-04-08
          Steven W. Reichenthal

-->

<!-- end of SRML 1.0 Schema .................................................. -->
<!-- ......................................................................... -->
```

12. Simulation Tools

Simulation requires an investment in tools, time, and people. Tools range from spreadsheets to programming languages to full commercial simulation packages. Each has tradeoffs. Spreadsheets, for example offer low purchase cost, but limited built-in functionality. Traditional programming languages provide maximum flexibility but can require more time and skill. General-purpose commercial simulation tools provide sophisticated built-in capabilities combined with graphical development environments that save time but can come with a high price tag. Commercial tools exist for specific applications such as supply chain management, resource management, manufacturing and science, making it easy to get a simulation running. In any case, a person who makes an investment in simulation should recognize some potential pitfalls:

- Spreadsheets and programming languages alone can require extensive programming, and spreadsheets can be too slow for complex situations where thousands of items need to be simulated.
- General-purpose tools can come with high price tags, and licenses can run into the tens of thousands of dollars per year.
- Tool vendors often go out of business or are purchased by larger firms who dissolve the product.
- The tools may not scale to handle highly sophisticated business processes.
- The programming languages underlying the tools can be obscure such that the learning curve becomes high, and only a few people will become skilled in their use.
- No widely adopted standards exist for interchanging simulations developed in different tools, making it difficult to reuse simulation models.
- Tools that support parallel processing or distribution are either not general-purpose, are very expensive, or require extremely advanced programming to use.

Without simulation support software, simulations developed in programming languages can lead to awkward programming. Even with a simulation engine, programming language solutions usually do not provide a natural means for describing hundreds or thousands of interconnected objects of various types.

13. Need for another markup language

SRML, like Dynamic HTML (DHTML) has scripting capabilities and object extensions that make it suitable for building simulations. However it's design was not intended for large-scale simulations. The following table compares the two:

Aspect	DHTML	SRML
Focus	Document-centric	Item-centric
Organization	Intertwined with a user interface and is difficult to separate.	SRML has no user interface
Data and objects	Allows for XML data islands for structured data, but DOM navigation methods are clumsy and the data has no intrinsic behavior.	Can individually specify behavior for every XML element via scripts.
Events	Pre-defined events happen on timers or user interaction, upward event bubbling mechanism.	Provides user-defined events that can be sent asynchronously, synchronously, or scheduled along with upward and downward event broadcasting..

14. References

[1] MathML is a standard under the World Wide Web Consortium (http://www.w3.org)